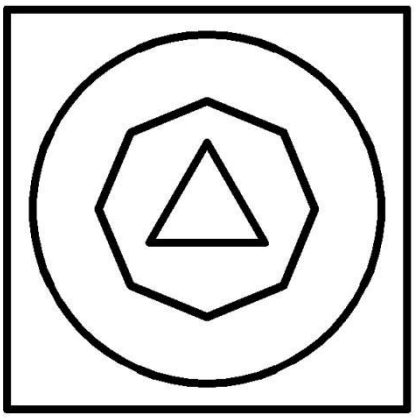
汇编杂谈



本文件由皇天惊虞制作，免费流通于网络。

制作时间　**2023.12.29**

LEA 就是目标地址传送指令： 将一个近地址指针写入到指定的寄存器。指令的功能是将源操作数、即存储单元的有效地址传送到目的操作数。

LEA 有两个操作数，分别为：

1、左边是目的操作数，表示操作结果保存在此，该指令目的操作数只能是 8 个通用寄存器之一；

2、右边的是源操作数，该指令的源操作数只能是一个存储单元，表达存储单元有多种寻址方式。

Lea 是返回偏移地址，对于 Lea EAX,s1 这样很好解释,就是返回变量 s1 的偏移地址

Lea EAX,[EBX+10]就比较复杂了

[EBX+10]表示 EBX+10 后作为地址,此地址指向位置的值,取的是 EBX+10 地址内的内容。

那 Lea 究竟是把这个内容作为地址存入 EAX，还是把这个内容的地址赋给 EAX 呢？如果是后者，那不是跟 MOV EAX,EBX+10 的效果一样了？

实际的效果应该是后者，Mov EAX,EBX+10 是非法的语句，汇编不允许操作数有 2 个寄存器和立即数存在。

Lea EAX,[EBX+10]的作用就是把 EBX+10 作为地址存入 EAX。

字节序，又称端序或尾序(英语中用单词：Endianness 表示)，在计算机领域中，指电脑内存中或在数字通信链路中，占用多个字节的数据的字节排列顺序。

在几乎所有的平台上，多字节对象都被存储为连续的字节序列。例如在 Go 语言中，一个类型为 int 的变量 x 地址为 0x100，那么其指针&x 的值为 0x100。且 x 的四个字节将被存储在内存的 0x100, 0x101, 0x102, 0x103 位置。

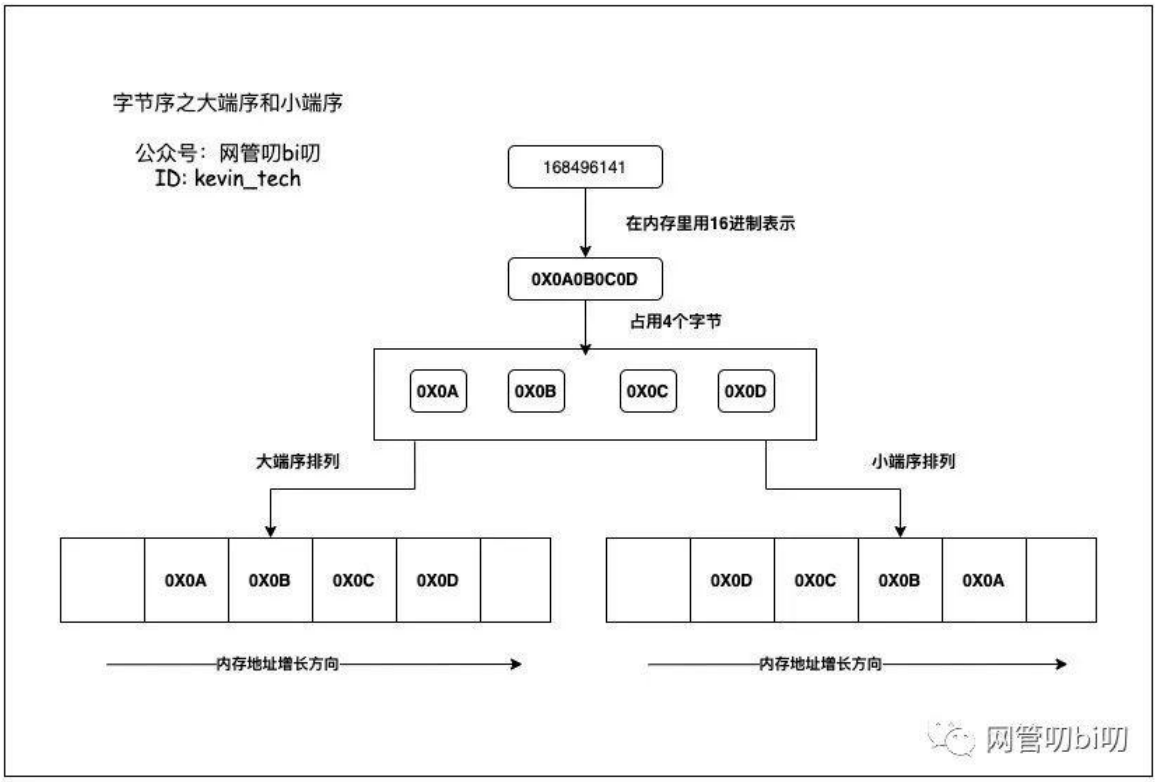
字节的排列方式有两个通用规则:

大端序(Big-Endian)将数据的低位字节存放在内存的高位地址，高位字节存放在低位地址。这种排列方式与数据用字节表示时的书写顺序一致，符合人类的阅读习惯。

小端序(Little-Endian)，将一个多位数的低位放在较小的地址处，高位放在较大的地址处，则称小端序。小端序与人类的阅读习惯相反，但更符合计算机读取内存的方式，因为 CPU 读取内存中的数据时，是从低地址向高地址方向进行读取的。

上面的文字描述有点抽象，我们拿一个例子来解释一下字节排列时的大端序和小端序。

在内存中存放整型数值 168496141 需要 4 个字节，这个数值的对应的 16 进制表示是 0X0A0B0C0D，这个数值在用大端序和小端序排列时的在内存中的示意图如下：



为何要有字节序

很多人会问，为什么会有字节序，统一用大端序不行吗？答案是，计算机电路先处理低位字节，效率比较高，因为计算都是从低位开始的。所以，计算机的内部处理都是小端字节序。在计算机内部，小端序被广泛应用于现代 CPU 内部存储数据；而在其他场景，比如网络传输和文件存储则使用大端序。

汇编结束不一定要带这两句，

```
mov    ax, 4c00h
```

int 21h

是 DOS 系统功能调 INT 21H 功能中的一种，表示带返回码结束用户程序。

不是定值，也可写成

```
mov    ah ,4ch
int 21h
```

int 21H 指令中，寄存器 AX，分为 AH 和 AL，AH 中存入指令码 4C 表示带返回码结束，AL=返回码。

AT&T 汇编与 Intel 汇编区别

X86 汇编语言有两种语法风格，即 Intel 汇编风格和 AT&T 汇编风格。其中 DOS 操作系统采用 Intel 汇编风格，而 Linux 内核源代码采用 AT&T 汇编风格。下面将简单介绍这两种风格的区别。

一、联系
Unix 系统最初是为了 PDP－11 而开发的，曾先后被移植到 VAX 及 68000 系列的处理器，这些处理器都采用 AT&T 汇编风格，所以 Unix 源代码中的汇编部分也采用 AT&T 汇编风格。当 Unix 被移植到 i386 处理器时，为了省事也采用了 AT&T 的汇编风格，而非 i386 处理器所希望的 Intel 汇编风格。Linux 作为 Unix 家族的一员，它的很多内容都发源于 Unix，所以 Linux 内核源代码也采用 AT&T 汇编风格。

尽管 AT&T 汇编语言与 Intel 汇编语言在语法上有一定的差异，但所基于的硬件知识是相同的，因此如果你熟悉 Intel 的语法格式，那么你也可以很容易地把它“移植“到 AT&T 来。

二、前缀
在 Intel 的语法中，寄存器和和立即数都没有前缀；但是在 AT&T 中，寄存器前冠以“％”，而立即数前冠以“$”。

在 Intel 的语法中，十六进制和二进制立即数后缀分别冠以“h”和“b”；而在 AT&T 中，十六进制立即数前冠以“0x”。

Intel 语法	AT&T 语法
mov eax,8	movl $8,%eax
mov ebx,0ffffh	movl $0xffff,%ebx
int 80h	int $0x80

三、操作数的方向
Intel 与 AT&T 操作数的方向正好相反。

在 Intel 语法中，第一个操作数是目的操作数，第二个操作数是源操作数。

在 AT&T 中，第一个数是源操作数，第二个数是目的操作数。

由此可以看出，AT&T 的语法符合人们通常的阅读习惯。

Intel 语法	AT&T 语法
mov eax,[ecx]	movl (%ecx),%eax

四、内存单元操作数
从上面的例子可以看出，内存操作数也有所不同。

在 Intel 的语法中，基寄存器用“[ ]”括起来；在 AT&T 中，基寄存器用“( )”括起来。

Intel	AT&T
mov eax,[ebx+5]	movl 5(%ebx),%eax

五、间接寻址方式
与 Intel 的语法比较，AT&T 间接寻址方式可能更晦涩难懂一些。

Intel 的指令格式是：

segreg:[base+index*scale+disp]
AT&T 的指令格式是：

%segreg:disp(base,index,scale)
其中 index/scale/disp/segreg 全部是可选的，完全可以简化掉。

如果没有指定 scale 而指定了 index，则 scale 的缺省值为 1。

segreg 段寄存器依赖于指令以及应用程序是运行在实模式还是保护模式下。在实模式下它依赖于指令，在保护模式下 segreg 是多余的。

在 AT&T 中，当立即数用在 scale/disp 中时，不应当在其前冠以“$”前缀。

Intel 语法

指令 segreg:[base+index*scale+disp]

AT&T 语法

指令 %segreg:disp(base,index,scale)

mov eax,[ebx+20h]	movl 0x20(%ebx),%eax
add eax,[ebx+ecx*2h	addl (%ebx,%ecx,0x2),%eax
lea eax,[ebx+ecx]	leal (%ebx,%ecx),%eax
sub eax,[ebx+ecx*4h-20h]	subl -0x20(%ebx,%ecx,0x4),%eax

从表中可以看出，AT&T 的语法比较晦涩难懂，因为[base+index*scale+disp]一眼就可以看出其含义，而disp(base,index,scale)则不可能做到这点。

这种寻址方式常常用在访问数组中某个特定元素内的一个字段，其中，base 为数组的起始地址，scale 为每个数组元素的大小，index 为下标。如果数组元素还是一个结构，则 disp 为具体字段在结构中的位移。

六、操作码的后缀
在 AT&T 的操作码后面有一个后缀，其含义就是指出操作码的大小。“l”表示长整数（32 位），“w”表

示字（16 位），“b”表示字节（8 位）。

在 Intel 的语法中，则要在内存单元操作数的前面加上 byte ptr、 word ptr 和 dword ptr，“dword”对应“long”。

Intel 语法	AT&T 语法
mov al,bl	movb %bl,%al
mov ax,bx	movw %bx,%ax
mov eax,ebx	movl %ebx,%eax
mov eax, dword ptr [ebx]	movl (%ebx),%eax

AT&T 汇编
Tom.Smile

1.语法格式

1.寄存器
引用寄存器要在寄存器号前加百分号%,如“movl %eax, %ebx

8 个 32-bit 寄存器 %eax，%ebx，%ecx，%edx，%edi，%esi，%ebp，%esp

8 个 16-bit 寄存器 它们事实上是上面 8 个 32-bit 寄存器的低 16 位：
%ax，%bx，%cx，%dx，%di，%si，%bp，%sp

8 个 8-bit 寄存器 %ah，%al，%bh，%bl，%ch，%cl，%dh，%dl
它们事实上是寄存器%ax，%bx，%cx，%dx 的高 8 位和低 8 位

6 个段寄存器 %cs(code)，%ds(data)，%ss(stack), %es，%fs，%gs

3 个控制寄存器 %cr0，%cr2，%cr3；

6 个 debug 寄存器 %db0，%db1，%db2，%db3，%db6，%db7；

2 个测试寄存器 %tr6，%tr7；

8 个浮点寄存器栈 %st(0)，%st(1)，%st(2)，%st(3)，%st(4)，%st(5)，%st(6)，%st(7)

寄存器功能
以 E 开头的寄存器为 32 位
EAX（累加器，是算术运算的主要寄存器）
EBX（基址寄存器，在内存中寻址时存放基址）
ECX（计数器）
EDX（数据寄存器）

ESI（源变址）
EDI（目标变址）
ESP（堆栈指针）
EBP（基址指针）
EIP（程序计数器，存储的是将要执行的下一条指令放在内存中的地址）
EFLAGS（保存的是根据运算得到的结果设置的条件码 ZF，CF，SF，OF）

段寄存器：
CS：代码段寄存器
SS：堆栈段寄存器
DS：数据段寄存器
ES、FS、GS：附加数据段寄存器

2.指令的长度后缀
在指令后面加上后缀表示长度，根据操作的是 1 字节、2 字节、4 字节、8 字节，分别对应后缀 b, w, l, q

3.操作数顺序
操作数排列是从源（左）到目的（右）
movl %eax(源）, %ebx(目的） (ebx) = (eax)

4.立即数
使用立即数，要在数前面加符号$
1:
movl $0x04, %ebx

2:
para = 0x04
movl $para, %ebx

5.注释
在 AT&T 中，注释既可以使用" ;“，也可以使用” !"

6.符号常数
符号常数直接引用

value: .long 0x12a3f2de
movl value , %ebx
注：指令执行的结果是将常数 0x12a3f2de 装入寄存器 ebx，引用符号地址在符号前加符号$, 如“movl $value, % ebx”则是将符号 value 的地址装入寄存器 ebx

7.调用和跳转指令
段内调用和跳转指令为"call"，“ret"和"jmp”，段间调用和跳转指令为"lcall"，“lret"和"ljmp”

段间调用和跳转指令的格式

lcall/ljmp $SECTION, $OFFSET

段间返回指令为
lret $STACK-ADJUST

jmp, jcc, call 指令跳转：

type	example	syntax address used
register	*%rax	contents of %rax
immediate	.l3	address of .l3
memory	*0x20(%rip)	value stored at address %rip + 0x20

寄存器：如果想要跳到寄存器%rax 内容对应的地址，不是写成 jmpq %rax，而是写成*jmpq %rax

立即数：不用加*号

内存：如果想要跳转到%rip + 0x10 对应的地址，不是写成 jmpq 0x10(%rip)，而是写成 jmpq *0x10(%rip)

8.数据声明

格式：命令 数据类型

.ascii 文本字符串
.asciz 以空字符串结尾的文本字符串
.byte 字节值
.double 双精度浮点数
.float 单精度浮点数
.int 32 位整数
.long 32 位整数（同 32）
.octa 16 字节整数
.quad 8 字节整数
.short 16 位整数
.single 单精度浮点数（和.float 同）

两个命令声明缓冲：

命令	描述
.comm	声明未初始化的数据的通用内存区域
.lcomm	声明未初始化的数据的本地通用内存区域

9.寻址方式
1.直接寻址
把某个地址上的值放到寄存器中
movl 0x8000, %eax # 把地址 0x8000 上的值放到 eax 中

2. 间址寻址

把寄存器上的值所代表的地址所指向的值放到寄存器中

```
movl $0x8000, %ebx
movl (%ebx), %eax   # 间址寻址，把地址 0x8000（在寄存器 ebx 中）上的值放到 eax 中
```

3. 基址寻址

以寄存器里的数值作为基址，加上一个常数得到最终地址，把地址上的值放到寄存器中

```
movl $0x8000, %eax
movl 4(%eax), %ebx   #基址寻址，把地址 0x8004（0x8000+4）上的值放到 eax 中
```

4. 变址寻址

以两个寄存器里的数值之和加上一个常数得到最终地址，把地址上的值放到寄存器中

```
movl $0x8000, %eax
movl $0x4, %ebx
movl （%eax，%ebx）, %ecx    #变址寻址，把地址 0x8004（0x8000+4）上的值放到 ecx 中
movl  4（%eax，%ebx）, %ecx    #变址寻址，把地址 0x8008（0x8000+4+4）上的值放到 ecx 中
```

5. 比例变址寻址

以一个寄存器里的数值加上另一个寄存器里的数字 乘以一个比例因子（1,2,4,8）再加上一个常数得到最终地址，把地址上的值放到寄存器中

```
movl $0x2000, %eax
movl $0x2, %ebx
movl （，%eax，4）, %ecx    #比例变址寻址，把地址 0x8000（0x2000 *4）上的值放到 ecx 中
movl  6（，%eax，4）, %ecx    #比例变址寻址，把地址 0x8006（0x2000 *4+6）上的值放到 ecx 中
movl  （%ebx，%eax，4）, %ecx    #变址寻址，把地址 0x8002（0x2000*4+2）上的值放到 ecx 中
movl  6（%ebx，%eax，4）, %ecx    #变址寻址，把地址 0x8008（0x2000*4+2+6）上的值放到 ecx 中
```

10.文件组成

.text：存放代码对应的指令 正文段

.bss：存放未初始化的全局和静态变量，在运行时该区域初始是全 0

.rodata：存放只读数据和变量，例如字符串字面量

.data：存放余下的数据和变量，可读可写 数据段

.ascii：定义一个字符串并且用双引号包含

.byte: 定义一个字符用单引号

.org:定义当前的汇编的位置

伪操作符语句是汇编器使用的指示符，它通常并不会产生任何代码，它由伪操作码和 0 个或多个操作数组成。每个操作码都是由一个一个点字符’.'开始，表示编译过程中的位置计数器。其值是点符号出现机器指

令第一个字节的地址。

11.编译

1）使用 as 命令对汇编文件进行汇编生成目标文件：as xxx.s -o xxx.o

2）使用 ld 命令对目标文件进行链接生成可执行文件：ld xxx.o -o xxx

注意，ld 命令进行链接要求目标文件的.text 段必须有一个入口点，ld 默认认为_start 标签对应的代码是入口点

12.操作码前缀
用于修饰随后的代码
操作码前缀 说明
cs, ds, ss, es, fs, gs 区覆盖操作码前缀，通过指定使用
区：内存操作数 内存引用形式会自动添加这种前缀
data16, addr16 操作数/地址宽度前缀。
会把 32 位操作数/地址改变为 16 位。（as 不支持 16 位寻址方式）
lock 总线锁存前缀。
用于指令执行期间禁止中断（仅对部分指令有效）
wait 协处理器指令前缀。
等待协处理器完成当前指令执行
rep, repe, repne 串指令操作前缀，重复执行 %ecx 中指定次数

word ptr 与 byte ptr

- `word ptr` 指明了指令访问的内存单元是一个字单元。
- `byte ptr` 指明了指令访问的内存单元是一个字节单元。
- `dword ptr` 指令访问的内存单元是一个双字单元。
- ```
  mov dword ptr fs:[0],ecx
  ```

ptr -- pointer （指针）缩写。

汇编里面 ptr 是规定的字 (保留字)。
  ```

mov ax,bx ;是把 BX 寄存器“里”的值赋予 AX，由于二者都是 word 型，不用 wordmov ax,word ptr [bx];是把内存地址等于“BX 寄存器的值”的地方所存放的数据，赋予 ax。

由于只是给出一个内存地址，不知道希望赋予 ax 的，是 byte 还是 word，所以需要用 word 明确指出！

例如，下面的指令中，用 word ptr 指明了指令访问的内存单元是一个字单元。
mov word ptr ds: [0],1
inc word ptr [bx]
inc word ptr ds: [o]
add word ptr [bx],2
下面的指令中，用 byte ptr 指明了指令访问的内存单元是一个字节单元。
mov byte ptr ds: [0],1
inc byte ptr [bx]
inc byte ptr ds: [o]
add byte ptr [bx],2

8086 汇编 PTR 的含义与用法 （十分全面详细）

el/2023/8/21 22:29:30
最近在学习汇编时对汇编代码里的 ptr 不是很清楚，而书上又没有详细的解释和例子，于是在网上看了些文章，整理总结一下。

ptr -- pointer （既指针）得缩写。

汇编里面 ptr 是规定 的 字 (既保留字)，是用来临时指定类型的。

（可以理解为，**ptr 是临时的类型转换，相当于 C 语言中的强制类型转换**）

如 mov ax,bx ； 是把 BX 寄存器“里”的值赋予 AX，由于二者都是寄存器，长度已定（word 型），所以没有必要加“WORD”

mov ax,word ptr [bx]; 是把内存地址等于“BX 寄存器的值”的地方所存放的数据，赋予 ax。由于只是给出一个内存地址，不知道希望赋予 ax 的，是 byte 还是 word，所以可以用 word 明确指出；如果不用，既（mov ax, [bx]; ）则在 8086 中是默认传递一个字，既两个字节给 ax。

总结，既有寄存器时可以，且一般不用 ptr；没有时一定要用（防止当两个操作数的宽度不一样）。

下面举几个具体点的例子，能帮助更好的理解：

（1）通过寄存器名指明要处理的数据的尺寸。（既有寄存器，可以不用 ptr 来限制了，系统会自动分析的）

例如：

下面的指令中，寄存器指明了指令进行的是字操作：

mov ax,1

mov bx,ds:[0]　　这个的意思是段内的偏移地址是 0，段地址是 DS。详情请看本人其他日记

mov ds,ax

mov ds:[0],ax

inc ax

add ax,1000

下面的指令中，寄存器指明了指令进行的是字节操作（因为是 al）：

mov al,1

mov al,bl

mov al,ds:[0]

mov ds:[0],al

inc al

add al,100

(2)在没有寄存器名存在的情况下，既都是在内存，得用操作符 X ptr 指明内存单元的长度，X 在汇编指令中可以为 byte，word 或者 DWORD。要不然内存是片连续的区域，操作就乱了。

例如：

下面的指令中，用 word ptr 指明了指令访问的内存单元是一个字单元：

mov word ptr ds:[0],1

inc word ptr [bx]

inc word ptr ds:[0]

add word ptr [bx],2

下面的指令中，用 byte ptr 指明了指令访问的内存单元是一个字节单元：

mov byte ptr ds:[0],1

inc byte ptr [bx]

inc byte ptr ds:[0]

add byte ptr [bx],2

在没有寄存器参与的内存单元访问指令中，用 word prt 或 byte ptr 显性地指明所要访问的内存单元的长度是很必要的。否则，CPU 无法得知所要访问的单元，还是字节单元。假如我们用 Debug 查看内存的结果如下：

2000：1000 FF FF FF FF FF FF ……

那么指令：

mov ax,2000H

mov ds,ax

mov byte ptr [1000H],1

将使内存中的内容变为：

2000: 1000 01 FF FF FF FF FF ……

而指令：

mov ax,2000H

mov ds,ax

mov word ptr [1000H],1

将使内存中的内容变为：

2000：1000 01 00 FF FF FF FF ……

这是因为 mov byte ptr [1000H],1 访问的是地址为 ds:1000H 的字节单元，修改的是 ds:1000H 单元的内容；而 mov word ptr [1000H],1 访问的是地址为 ds:1000H 的字单元，修改的是 ds:1000H 和 ds:1001H 两个单元的内容。

（3） 其他方法

有些指令默认了访问的是字单元还是字节单元，比如：push [1000H] 就不用指明访问的是字单元还是字节单元，因为 push 指令只进行字操作。

```
data

mydouble    dword    12345678h

.code

mov ax, mydouble ;这是不允许的，因为 mydouble 被声明为 4 字节元素，ax 寄存器只能存放 2 个字节。属于左右操作数不一致的情况。

mov ax, word ptr mydouble ;虽然 mydouble 是 4 字节元素，但 word ptr 表示从 mydouble 中只提取 2 个字节放入 ax。ax=5678，因为是小端模式，mydouble 的起始位置第一个字节是 78，第二个字节是 56。

mov ax, word ptr mydouble+1 ;ax=3456

mov ax, word ptr mydouble+2 ;ax=1234

mov ax, word ptr mydouble+3 ;超出了 mydouble 数据的范围一个字节，ax 的低位是 12，高位未知。
PTR 必须与一个标准汇编数据类型一起使用，包括：byte, sbyte, word, sword, dword, sdword, fword, qword, tbyte。

.data
mydouble word 1234h, 5678h

.code
main PROC
    mov eax, 0
    mov eax, dword ptr mydouble    ;mydouble 被定义为 2 字节元素，左右操作数必须同样大小的原则，用
                                   dword ptr 来提取 4 字节才能交给 eax 寄存器。
    invoke exitprocess, 0
main ENDP
END main
```

指令要处理的数据有多长

8086CPU 的指令，可以处理两种尺寸的数据，byte 和 word。所以在机器指令中要指明，指令进行的是字操作还是字节操作。

1）通过寄存器名指明要处理的数据的尺寸。

字操作：

```
mov ax, 1
```

```
mov bx, ds:[0]


mov ds, ax


mov ds:[0], ax


inc ax


add ax, 1000
```

字节操作：

```
mov al, 1


mov al, bl


mov al, ds:[0]


mov ds:[0], al


inc al


add al, 100
```

2）在没有寄存器名的情况下，用操作符“X ptr”指明内存单元的长度，X 在汇编指令中可以为 word 或 byte。

用 **word ptr** 指明了指令访问的内存单元是一个字单元。

```
mov word ptr ds:[0], 1


```

```
inc word ptr [bx]


inc word ptr ds:[0]


add word ptr [bx], 2
```

用 **byte ptr** 指明了指令访问的内存单元是一个字节单元。

```
mov byte ptr ds:[0], 1


inc byte ptr [bx]


inc byte ptr ds:[0]


add byte ptr [bx], 2
```

在没有寄存器参与的内存单元访问指令中，用 **word ptr** 或 **byte ptr** 显性地指明所要访问的内存单元的长度是很有必要的。否则，CPU 无法得知所要访问的单元是字单元，还是字节单元。

例：

内存：

```
2000：1000 FF

2000: 1001 FF

2000: 1002 FF
```

指令：

```
mov ax, 2000H

mov ds, ax

mov byte ptr ds:[1000H], 1
```

则地址变为：

```
2000：1000 01

2000: 1001 FF

2000: 1002 FF
```

“**mov byte ptr ds:[1000H], 1**”访问的地址为 **ds:1000H** 的字节单元，修改的是 **ds:1000H** 单元的内容。

指令：

```
mov ax, 2000H

mov ds, ax

mov word ptr ds:[1000H], 1
```

则地址变为：

```
2000：1000 01

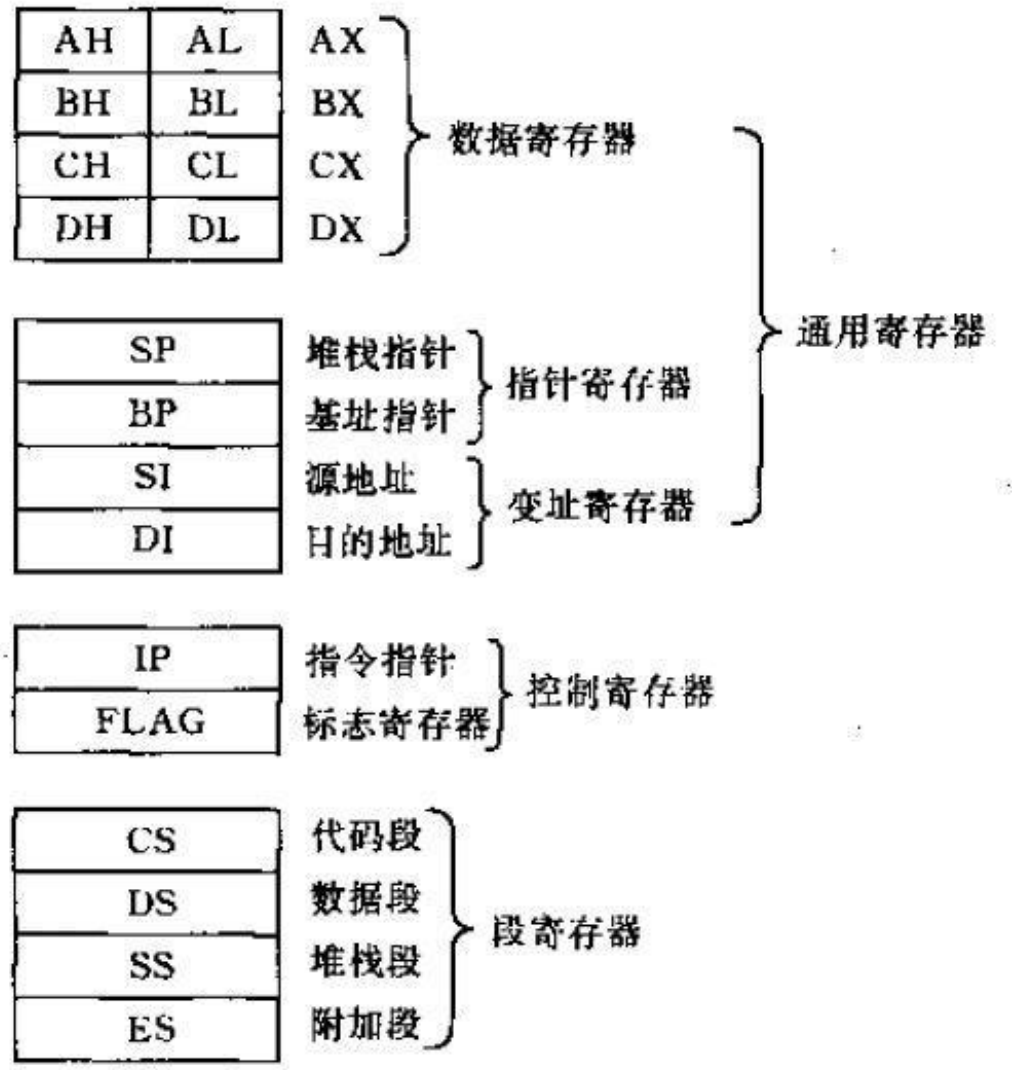
2000: 1001 00

2000: 1002 FF
```

“**mov word ptr ds:[1000H], 1**”访问的地址为 **ds:1000H** 的字单元，修改的是 **ds:1000H** 和 **ds:1001H** 两个单元的内容。（此时 **ds:1000H** 是低位，**ds:1001H** 是高位）

3）其他方法

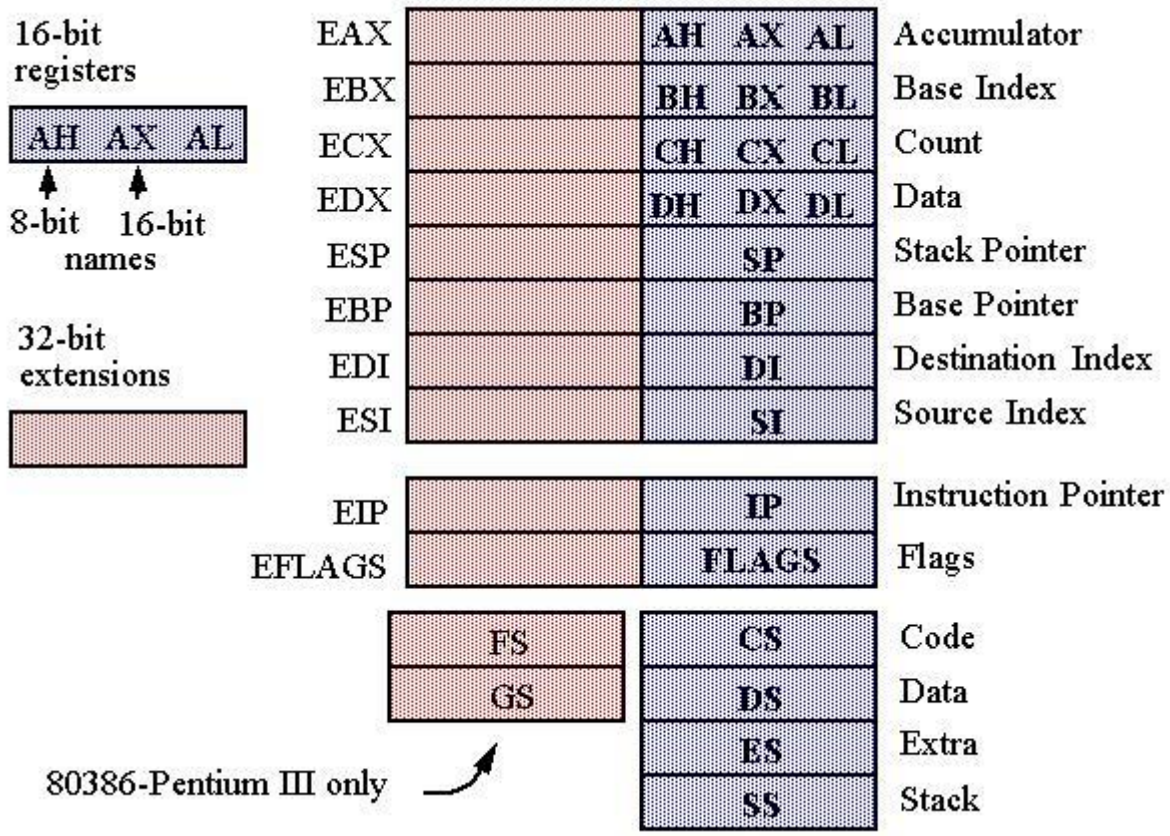
有些指令默认了访问的是字单元还是字节单元，比如，**push[1000H]**就不用指明访问的是字单元还是字节单元，因为 **push** 指令只进行 **字操作**。



C 语言中 80386 寄存器的基本用法总结

80386 寄存器共有 34 个寄存器，可分为 7 类，它们是通用寄存器，指令指针和标志寄存器，段寄存器，系统地址寄存器，控制寄存器、调试和测试寄存器。我们经常碰到的是前四类寄存器，这也是我这篇文章总结的重点。

通用寄存器



80386 寄存器有 8 个 32 位通用寄存器这 8 个通用寄存器都是由 8088/8086/80286 的相应的 16 位通用寄存器扩展成 32 位而得。名字分别是：EAX，EBX，ECX，EDX，ESI，EDI，EBP，ESP。每个 32 位的通用寄存器的最第 16 位可以单独使用，对应于 8088/8086/80286 的相应 16 位通用寄存器作用相同。同时，EAX，EBX，ECX，EDX 四个寄存器的低 16 位 AX,BX,CX,DX 还可以继续分为各高 8 位，低 8 位寄存器单独使用，例如 AX 可以分为 AH 和 AL，每个都是 8 位寄存器。

这 8 个为通用寄存器，说明它们的用处不止一个。它们通常既可以保存逻辑和算术运算中的那个操作数，也可以保存地址运算中的操作数。
下面介绍一下在 c 程序和系统调用时，这些寄存器的用法:

```
#include<stdio.h>
int sum(int x, int y)
{
   int accum = 0;
   int t;
   t = x + y;
   accum += t;
   return accum;
}

int main( int argc, char **argv)
{
   int x = 1, y = 2;
   int result = sum( x, y );
```

```
    printf("\nresult = %d\n", result);
    return 0;
}
```

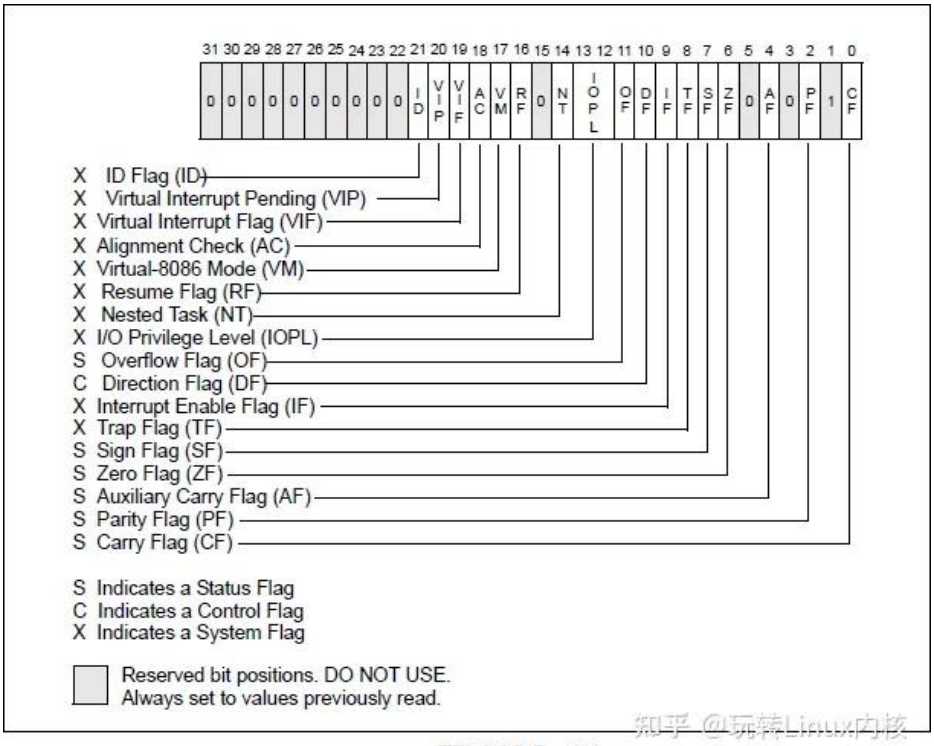
在函数调用和系统调用时，需要先将参数压入栈，然后被调用函数再从相应的寄存器获取参数值，存储在被调用函数的栈中，所以被调用函数对参数做出的改变并不会修改主函数的数值，因为他们在不同的栈中。函数的参数存入栈的顺序是从右到左，如果是调用函数 sum(x,y),则是先将 y 压入栈，然后再将 x 压入栈。
对于寄存器而言，**c 程序和系统调用的参数则必须按顺序放到寄存器 ebx，ecx，edx，esi，edi 中，而函数调用的参数操作是从右到左，所以最后一个参数放入 esi，倒数第二个参数存入 edi 等以此类推。**对于上述的例子 main 函数调用 sum(x,y)，main 函数将 y 存入 esi，x 存入 edi。然后 sum 函数内部从 esi 和 edi 获取参数值。

而 eax 在函数调用中，经常用来存储函数的返回值。上述例子中，sum 函数将值存入 eax 寄存器中，main 函数再从 eax 寄存器获取返回值。对于 fork 函数返回两个值，也可以通过 eax 来理解，当内核调度父进程时，键 pid 存入 eax 寄存器，当内核调度子进程时，将 0 存入 eax 寄存器。
esp 寄存器为栈顶寄存器，并且始终指向栈顶。

ebp 寄存器为基址寄存器，其实就是每个函数栈的栈的寄存器，通过这个基址寄存器，再加上偏移量，即可获取参数以及局部变量的值。

指令指针寄存器和标志寄存器

EIP 为指令指针寄存器，是 32 位寄存器，低 16 位称为 IP，用于兼容 16 为 CPU，其内容是下一条要取入 CPU 的指令在内存中的偏移地址。当程序刚运行时，系统把 EIP 清零，每取入一条指令，EIP 自动增加相应的字节数，指向下一条指令。



EFLAGS Register

EFLAGS 也是为标志寄存器，低 16 位称为 FLAGS，与 16 位 CPU 的标志寄存器同名，同作用。可分为 3 类：状态标志，控制标志和系统标志，简述如下：

AF——辅助进位标志。若该位置位时，表示最低有效的 4 位向高位产生了进位或借位，则该标志位主要用于 BCD 算术运算。
CF——进位标志。当该位置位，表示 8 位或 16 位或 32 位数的算术操作产生了进位或借位。进行多字节数的加、减时要使用该标志。循环移位指令也影响进位标志。
PF——奇偶标志。主要用于数据通讯应用程序中，当该位置位时，表示结果数据位中有偶数个 1，可以检查数据传送中是否出现错误。
SF——符号标志。该位置时表示结果的最高位(符号位)为 1。对于带符号数，该位为 1 表示负数，该位为 0 表示正数。
ZF——零标志。当该位置位时，表示操作的结果为 0。
DF——方向标志。用于控制数据串操作指令中的地址变化方向。DF 为 0 时，SI/DI 或 ESI/EDI 为自动增量，地址从低向高变化，DF 为 1，SI/DI 或 ESI/EDI 为自动减量，地址从高向低变化。
IF——中断允许标志。该位置 1 时允许响应外部可屏蔽中断(INTR)，该位复位时禁止响应外部可屏蔽中断。IF 不影响非屏蔽外部中断(NMI)或内部产生的中断。
OF——溢出标志。若该位置位表示此次运算发生了溢出，即作为带符号数运算，其结果值超出目的单位所能表示的数值范围。这时目的单位的内容对带符号数没有意义。
TF——陷阱标志。当该位置位时，把处理器置成供调试的单步方式。在这种方式中，每条指令执行后 CPU 自动产生一个内部中断，使调试者可以观察程序中该条指令执行的情况。
NT——嵌套任务标志。用来表示当前的任务是否嵌套在另一任务内，当该位置 1 时，表示当前的任务有一个有效的链连接到前一个任务(被嵌套)，如果执行 IRET 指令，则转换到前一个任务。
IOPL——输入/输出特权级标志，用于定义允许执行输入/输出指令的 I/O 特权级的数值。
RF——恢复标志。它是与调试寄存器的断点一起使用的标志，当该位置 1 时，即使遇到断点或调试故障，也不产生异常中断 1。在成功地执行每条指令时，RF 将自动复位。
VM——虚拟 8086 方式标志。当该位置位时，CPU 工作在虚拟 8086 模式(简称为拟 86 模式)，在这种模式下运行 8086 的程序就好象是在 8086CPU 上运行一样。
AC——对准检查标志。这是 80486 新定义的标志位。该位置时，如果进行未对准的地址访问，则产生异常中断 17。所谓未对准的地址访问，是指访问字数据时为奇地址，访问双字数据时不是 4 的倍数地址，访问 8 字节数据时，不是 8 的倍数的地址。对准检查在特权级为 0，1，2 时无效，只有在特权级 3 时有效。
s—状态标志；c—控制标志；x—系统标

段寄存器

可能在刚接触汇编时，会觉得段寄存器存储的就是程序每个段的基地址，在了解了 linux 虚拟地址转为物理地址之后，我才知道原来段描述符存储的并不是段基地址，而是存储了在段描述符表的索引以及一些属性。80386 有 6 个段寄存器，分别是 CS，DS，SS，ES，FS 和 GS，是 16 位寄存器。前 4 个段寄存器的名称与 8088/8086 相同，在实地址方式下使用方式也和 8088/8086 相同。80386 又增加了 FS 与 GS，主要为了减轻对 DS 段和 ES 段的压力。

这些 16 位段寄存器存放的并不是段基地址，而是存储了在段描述符表的索引，D3-D15 位是索引值,D0-D1 位是优先级(RPL)用于特权检查,D2 位是描述符表引用指示位 TI,TI=0 指示从全局描述表 GDT 中读取描述符，TI=1 指示从局部描述符中 LDT 中读取描述符。这些信息总称段选择器(段选择器).
系统地址寄存器

这些寄存器是我写这篇文章最主要的原因，因为这些寄存器在汇编代码中很少见，而在理解地址转换时又是那么重要。

系统地址寄存器有四个，用来存储操作系统需要的保护信息和地址转换表信息、定义目前正在执行任务的环境、地址空间和中断向量空间。

GDTR 48 位全局描述符表寄存器，用于保存全局描述符表的 32 位基地址和全局描述符表的 16 位界限（全局描述符表最大为 216216 字节，共 216/8=8K216/8=8K 个全局描述符）。GDT 表里面的每一项都表明一个段的信息，或者是一个 LDT 表的相关信息。其实一个 LDT 表也是一个段。所以也可以说 GDT 表的每一项都描述一个段。就像一个文件夹下面可以有文件，也可以有文件夹一样，GDT 表里面既可以有段描述符，也可以有 LDT 的表。

IDTR 48 位中断描述符表寄存器，用于保存中断描述符表的 32 位基地址和中断描述符表的 16 位界限（中断描述符表最大为 216216 字节，共 216/8=8K216/8=8K 个中断描述符）。

LDTR 16 位局部描述符表寄存器，用于保存局部描述符表的选择符。一旦 16 位的选择符（也叫选择子）放入 LDTR，CPU 会自动将选择符所指定的局部描述符装入 64 位的局部描述符寄存器中。

TR 16 位任务状态段寄存器，用于保存任务状态段(TSS)的 16 位选择符。与 LDTR 类似，一旦 16 位的选择符放入 TR，CPU 会自动将该选择符所指定的任务描述符装入 64 位的任务描述符寄存器中。注：TSS 是一个段，所以在 GDT 中有对应的表项描述。

所以 LDTR 和 TR 寄存器分别是局部描述符表的选择符和任务状态段选择符，它们需要到 GDT 表格中找到各自的描述符才能定位到各自的表格，所以可以把这两个寄存器看作是和 cs,ds 等段寄存器一样，局部描述符表和任务状态段看作是一个段。